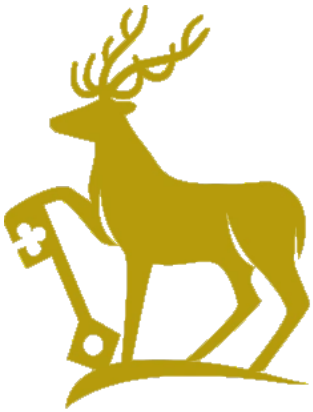

UNIVERSITY OF SURREY

EEE3027: Calculator Assignment

Assignment Report

2nd of May 2023

URN: 6596386

Department of Electronic Engineering
Faculty of Engineering and Physical Sciences
University of Surrey

Abstract

This report covers the work done on creating and testing an FPGA Calculator in VHDL as part of EEE3027 Digital Design with VHDL. VHDL was used to program an FPGA capable of taking serial input, converting the serial input to numbers and operations, calculating the result, converting the result back into serial, and finally transmitting the result back. The final design allows for a user to input, over serial, any integer between -32768 and 32767 (a signed 16-bit value) with the following integer operations: addition, subtraction, multiplication, and division. The FPGA will then return the result if it can be represented as a signed 16-bit value, otherwise it will return an error. The report also covers background theory for UART communication, an exploration of the original provided code, and a discussion on timing and physical implementation on a Minized Zynq board.

Contents

1 Overview

The objectives of this assignment were to create an arithmetic calculator using the std.numeric library and the provided UART code. This provided code first has to be debugged and tested. The calculator's basic requirement was to compute values given in the "$A+-*/B$" format, with no specific instructions of the range of values of A, B or the output. For the implementation created for this assignment, the calculator can take any input and provide any output that can be described as signed 16-bit integers (ie, between -32768 and 32767), however, through the use of generics, this range can be changed. The serial UART input is then processed into the two expected numbers and operations, with any character that is not a numeral (0 to 9) or an operator character ($+-*/$) being ignored. When any value is received, it will be echoed back via the transmitter. At the end of the input, a carriage return character (ASCII Hex code $0D$) is required to indicate the end of the number B. A carriage return is also transmitted after the result's final digit. Furthermore, the final design outputs a special character, "!", when an operation is invalid (such as dividing by zero), or the output is outside the calculator's range.

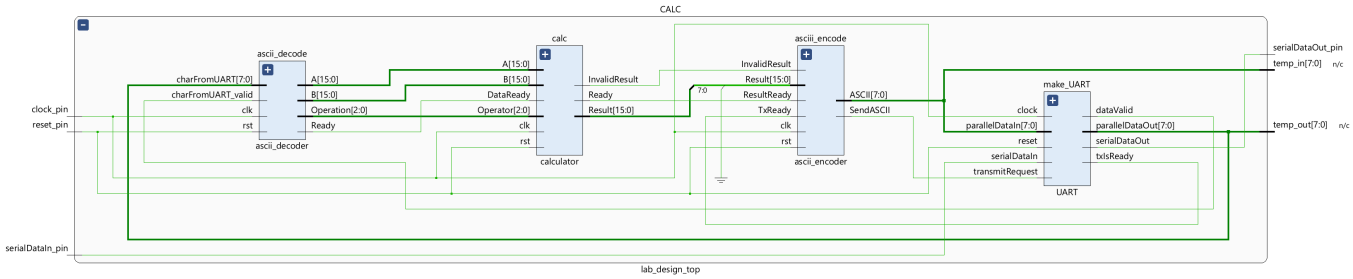

Figure 1: Top Level Implementation of Calculator

To achieve this functionality, the design seen in Figure 1 was created. This design shows four main components all interacting together. These components are the UART block, the ASCII decoder, the calculator, and the ASCII encoder. The UART block handles the receiving of serial UART data, converting it to a byte of parallel data as well as taking in parallel data and transmitting it as serial UART data. Anytime a full byte comes into the UART block's receiver, it will echo that byte back via the transmitter. The ASCII decoder takes parallel data and then, upon receiving valid ASCII codes, converts sequences of parallel data into two numbers and an operation. The calculator computes an operation between two numbers and then provides a result. It also has error checking and will signal if the operation is invalid. The ASCII encoder takes an integer and the invalid signal and converts that information into a sequence of parallel ASCII codes. Once converted, the codes are sent to the UART block's transmitter. Details on the design and operation of all of these components as well as information on test benches and implementation on a Minized Zynq FPGA board are provided in this report.

2 UART

2.1 UART Theory

UART stands for universal asynchronous receiver-transmitter and is a very common device-to-device communication protocol[1]. For this assignment, a two-way UART connection is utilized, allowing two devices (the FPGA and a connected PC) to transfer data between each other regardless of clock rate as long as they communicate over a predefined rate known as the baud rate. Unlike other communication protocols such as SPI or I2C, UART only ever has one "master" device and one "slave" device. A third device can never be added on the same lines.

UART takes parallel data and converts it into a bit-by-bit frame to send. The frame always starts with the start bit, which takes the line low for one cycle, as a UART line is usually high. AFter this, each data bit is sent one by one (least significant bit first), with a one being high and a zero being low. Some implementations will include a parity bit after the data, which will be zero if the data total is odd and one if the total is even. Parity is helpful to ensure that data has not been altered during transmission, which can

occur due to "electromagnetic radiation, mismatched baud rates, or long-distance data transfers"[1]. The provided UART, however, does not utilize the parity bit. At the end of the message, a stop bit is sent by asserting from low to high. Equation 1 shows an example of how data gets converted to a UART packet.

$$\text{Data} = 01000001 \tag{1}$$

$$\text{UART Frame} = \overset{\text{Start Bit}}{\{0\}} \quad \underset{\text{Data}}{\{10000010\}} \quad \overset{\text{Stop Bit}}{\{1\}}$$

The final key concept behind UART is the baud rate, which is the number of signals per second [2]. The UART baud rate represents how long each bit is held high or low. As UART is asynchronous, each device needs to know the baud rate ahead of time and generate its clock at the baud rate. This is enough to transmit UART as the serial output line only needs to be updated once each clock cycle, however for receiving, the line needs to be checked more often as the devices baud clocks can be out of phase by half a period and because real signals have non-instantaneous transition periods from high to low meaning that is best to check the signal in the middle of each cycle. To achieve this goal, a secondary oversampled clock is utilized. This clock is often by a factor of 16. This oversampled clock is used in the receiver to check the line in the middle of each received bit instead. This is done by waiting for half a baud rate after a start bit is received to sample the next bit. After this point, the receiver will wait for 16 oversampled pulses to take the following sample. Some implementations may take multiple samples around the midpoint to get a more accurate result; this implementation takes only one measurement. Once enough measurements have been made and the receiver has received the stop bit, it will assert high on the ready line with the data ready on eight parallel lines.

2.2 UART Component

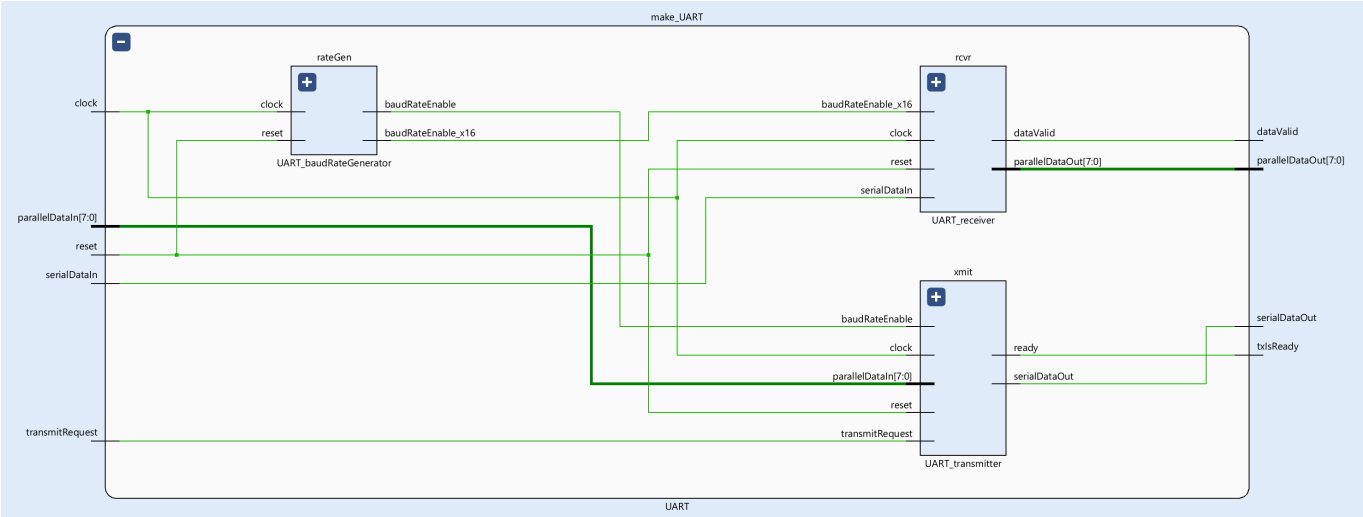

Figure 2: UART Block Diagram

The VHDL code for a UART block used in this assignment was provided. A few corrections were required to get the code in working order. The simpler of the fixes was that all files required the addition of the IEEE std_logic_1164 library. Testing also revealed that the UART receiver was not connected to the baudRateEnable_x16 line, which was quickly fixed in the port map. With these fixes, the components functioned as expected.

The baud rate generator, once given a clock line, a predefined clock frequency, and a baud rate, will produce two outputs: a pulse at the baud rate and a pulse at 16 times the baud rate, which connect to the transmitter and receiver respectively. It does this by counting the number of clock cycles and sending

a pulse every x cycles. Where x is the clock rate over the baud rate for the main pulse and one-sixteenth of that number for the oversampled pulse. This method means that the actual baud rate has a resolution limited by the clock rate.

The baud rate transmitter takes five inputs: a clock, a reset, baud pulses, and a transmit request line, as well as an 8-bit bus for the input data. Every clock cycle, the transmitter checks if the transmit request is high. Once it is high, the transmitter sends a UART message, including the start and stop bit, containing the data on the parallel input of the UART block. This operation is achieved using a finite state machine, a diagram for which can be seen in Figure 3

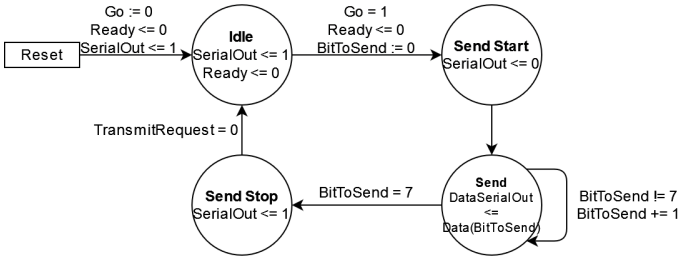

Figure 3: UART Transmitter State Machine Diagram

This diagram shows all the states, transitions, and conditions for transitions. The outputs in each of those states and transitions are also shown. As the output depends on the states and inputs, this is known as a Mealy state machine [3]. The = sign indicates a condition, with <= being used when signals are set and := when variables are being set. This format is used throughout the report. Note that for this component, the transitions will occur only on the rising edge of baud pulses, not during the rising edge of clocks. The exception is the transmit request, which is monitored on the clock rising edge and stored in a buffer as go.

The UART receiver takes in only a clock, a reset, and the oversampled baud pulses. As an output, it will give 8 bits of parallel data and a signal when the data is ready. This operation is also achieved with a state machine, seen in Figure 4

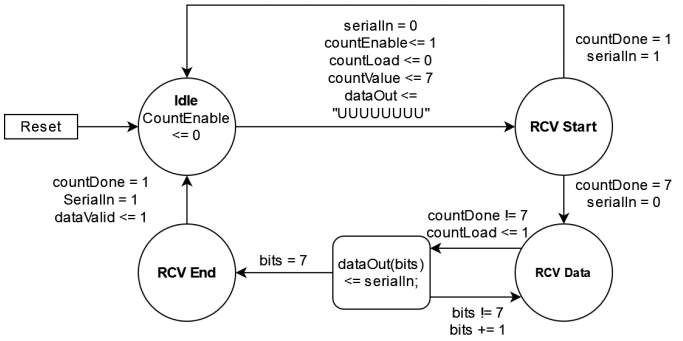

Figure 4: UART Receiver State Machine Diagram

The core operation is similar; however, as mentioned in the theory discussion, the receiver uses a 16 times oversampled baud rate to take samples in the middle of each incoming bit. The receiver keeps an internal count of the amount of oversampled pulses receiver after the start bit is detected in order to take a sample in the middle of each bit, Once this count is at 7, or roughly halfway into the start bit, the receiver measures to ensure the start bit was indeed received. If the start bit is correctly detected at this measurement, the receiver will wait for another 16 counts to measure in the middle of the first bit, and it will continue doing this until the stop bit is detected.

2.3 Test Bench

To test the UART module, a test bench was created with the two instances of the UART module. The transmitter in the first UART module (UUT_1) was given parallel data from a text file. The text file

contains hex codes, and the test bench will send them one by one to the transmitter, waiting each time for the transmitter's ready line to go high first. The serial output from this transmitter is then sent to the serial input of the receiver in UUT_2, whose output is then compared to the value read from the file to confirm everything is working. Using the modules like this allowed for testing the operability of the module with different clock rates very quickly, as the baud rate generators of UUT_1 and UUT_2 could easily be connected to clocks with different clock rates. This testing confirmed that transmission and reception of every 8-bit value work correctly after creating an input file with hex values $0x00$ to $0xFF$. Testing was repeated at different clock rates for both the receiver and transmitter (1.8432 MHz, 8 MHz, 20 Mhz, 50 Mhz), which all functioned in simulation. Some of these values, however, are unlikely to work with actual clocks as they would need to be extremely accurate clocks to generate acceptable baud pulses.

3 Original Design

Once the UART was tested, the remainder of the original top-level design was investigated. The design needed minor fixes and was cleaned up to use proper spacing. This original design would take in any serial inputs, convert them with the receiver, and check for three special cases in the decoder: if the data is between $0x61$ to $0x7A$ (a lowercase letter), if the data is between $0x41$ to $0x5A$ (an uppercase letter), or if the data is between $0x31$ to $0x39$ (a numeral). For lowercase letters, the decoder will subtract 32 before sending the value on, converting them to uppercase. For uppercase letters, the decoder will add 32 before sending, converting them into lowercase. The data would be sent unaltered for every other case, including numerals. These special cases also control the LEDs. When a letter is decoded, the lo_LED will go high, and the hi_LED will go low. When a numeral is decoded, the lo_LED will go low, and the hi_LED will go high. In all other cases, both LEDs will go high.

The decoder's data then goes to the encoder, which will send the character from the decoder to the UART block's transmitter if one is received. Otherwise, it will send a character based on the DIP switches. These switches, however, are now obsolete along with the debouncers connected to them as the new FPGA board does not support them. They used to serve to pick the value sent by the transmitter when the receiver received no new character.

4 UART Calculator

The original design was heavily modified to meet the requirements for this assignment. This involved removing the original encoder and drastically modifying the decoder. Figure 1 shows this modified design. As the UART components remained unchanged, it will not be discussed again. The design has three inputs: the clock, a reset line, and the UART in. There is only one output: the UART out. There are four generics: the clock rate, the baud rate, the maximum integer value, and the minimum integer value. The first two values are used only by the UART component so that the baud pulses can be appropriately generated. The integer range will be used to dynamically create versions of the calculator cable for handling larger integers. All integer registers in the design use these generics to set their limits, and the needed amount of registers for the output digits are created using a synthesis-only log10ceil function. In practice, the limit is -32768 to 32767 (a 16-bit signed integer); this is because the methods used to create different size calculators rely on using a loop during synthesis and Vivado limits loops to 2^{16}.

4.1 ASCII Decoder

The first step in computing a calculation is to take the ASCII inputs, already converted from serial to parallel in the UART receiver, and further process them into two numbers, A and B, and an operation. This conversion occurs in the ASCII decoder. The first step is determining what each ASCII represents. The decoder does this by converting the parallel data to an unsigned integer and then checking if the integer is between 48 to 71. These decimal values represent the ASCII characters for numerals 0 to 9. If so, the decoder outputs a signal labeling the message as a numeral (a custom type created for the calculator). It also outputs the numeral as a 4-bit number (subtracting 48 from the incoming data). Otherwise, the

decoder checks for operations $(+ - */)$, which are ASCII codes 43, 45, 42, and 57. If these are detected, the decoder outputs a signal labeling the incoming data as an operator (the same custom type as before) and another custom type marking the specific operator. The final ASCII character that is checked for is number 13, which is a carriage return. For this design, a carriage return is used to signify the end of a message. With any valid message, the decoder will also output a "character ready" signal for the state machine. If the message is any other value it is ignored, meaning that an input of $-AB1cs2 + 3hh4$ will be treated as $-12 + 34$.

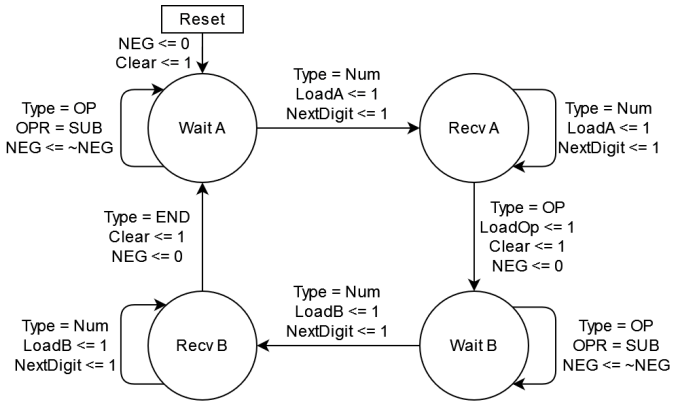

Figure 5: ASCII Decoder Mealy State Machine Diagram

Now that the decoder understands the incoming data, the individual numerals and operators must be interpreted. Interpretation is done with the state machine (SM) in Figure 5. The SM will check for its transition conditions whenever any inputs change. In its initial state, Wait for A, it will wait to receive either a valid numeral or the negative operator. With the latter, the SM will mark the upcoming value as negative; this can occur multiple times meaning that $---1$ will be correctly interpreted as -1. Once a numeral is received, the SM signals the digit-to-number converter to read the value, and the SM will transition to the Receiving A state. In this state, the SM checks for new numerals and will again signal the digit-to-number converter when this is the case. Once any operation is received, the SM will mark the end of number A and transition to the Waiting for B state. The operation will also be stored in the operation (op) register. The Waiting for B and Receiving B states mirror the A states. However, instead of looking for an operation to mark the end of number B the carriage return is awaited. In both cases, the digit-to-number converter works by storing the current value, which starts at 0, and upon receiving a new number multiplying the current value by ten and then adding the new numeral to the current value. This new value is made negative if the SM outputs the negative signal. The registers will store this value when the SM outputs the load A or load B signal at the end of Receiving A and Receiving B states. These load signals will also clear the converter's current value to allow it to convert the next series of numerals. Finally, when the SM moves from Receiving B back to Waiting for A, it will output a ready signal, indicating that register A, B, and Op contain all the data that needs to be computed.

To achieve this conversion, the numeric STD library is utilized in the decoder with the ASCII data and in the digit-to-number converter turning numerals into larger integers.

4.2 Calculator

Despite being a calculator, the calculating component of this assignment is one of the simplest. This simplicity is thanks to the numeric STD library, which implements the logic required for adding, subtracting, multiplying, and dividing integers. Furthermore, as UART communication is orders of magnitude slower than the clock frequency (115.2 kHz vs 50 MHz) the speed of even the slowest calculations appears near instant compared to communication (this can be seen on the simulation graph in Figure 9).

The calculator takes in a clock, two integers, an operation, and a ready signal as input. These inputs are then used to produce an integer and a signal indicating that the calculator is done. The calculator also has an extra signal indicating an invalid operation, which occurs when dividing by zero or exceeding the signed integer limit. The current implementation of the calculator is state machine driven, shown in

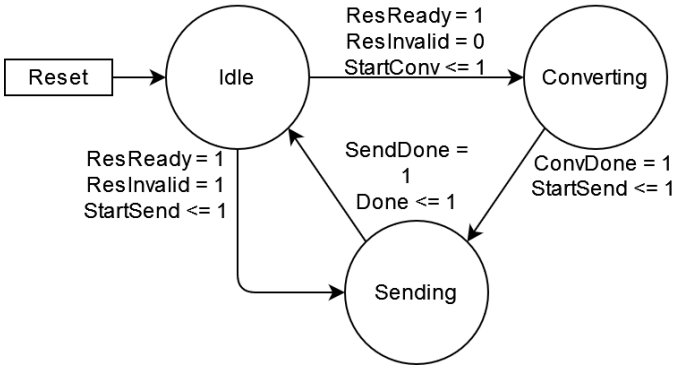

Figure 7: ASCII Encoder Mealy State Machine Diagram

Figure 6. Upon being informed that the data and operation are ready the SM will send a start signal to the appropriate sub-module for each operation. Then it will wait for the component to send a ready signal, at which point the calculator will forward the result and invalid line from the sub-module to the next encoder along with a done signal.

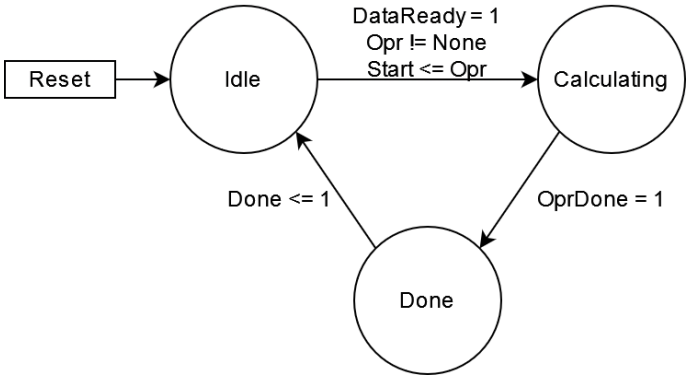

Figure 6: ASCII Calculator Mealy State Machine Diagram

Each sub-module is similar in design, with all four of them awaiting a start signal which will trigger the operation to begin. Before and after the operation, some error checking is done. The error checking is specific to each module, with the divider checking for divide-by-zero errors and the others checking if the new output is below the minimum value or above the maximum value. Once the result is done, valid or invalid, the sub-module will output a done signal.

This approach is inefficient, as each module is independent and shares no components with other components. As two operations will never occur simultaneously, this could be made more efficient. However, the current design is simple and allows for easy expansion, such as adding an exponent function to the calculator.

4.3 ASCII Encoder

Once an integer result is calculated, the result needs to be converted into a series ASCII characters and sent out one by one to the UART transmitter. These operations occur in the ASCII Encoder and Sender. This component takes in a clock, a reset, a signed 16-bit integer result, a done signal from the calculator, and the transmitter's ready signal as input. The outputs are 8 bits of output data and a send signal. The conversion and sending stages happen sequentially, meaning that a simple state machine is used to control the encoder, shown in Figure 7

When the result is ready, the state will transition to converting if the result is valid or go straight to sending if the result is invalid. Converting from numbers back to digits is more challenging than the inverse as rather than storing one integer, multiple numerals need to be stored. To store all the possible numerals registers are created on synthesis based on the given integer limit. In the case of the signed 16-bit number,

the largest value is 32767, so five registers are required. These register's loading is done using a number-to-digit converter and a counter. The converter starts by taking the remainder of the result when divided by ten. This remainder is then stored in the register with the index of the counter's output. The result is then divided by ten. Finally, if the newly divided result can be divided by ten again and not be zero, the counter is incremented and the process starts again. Following this process means that any number will have its digits stored in the registers, as 4-bit values, with the counter indexing the highest-order term. This process is identical for negative and positive values, as the only difference will be whether a negative sign will be sent, which will be handled by the sender. Once the conversion is done, a ready signal is sent and the encoder's state machine will move to the sender.

4.3.1 Sender

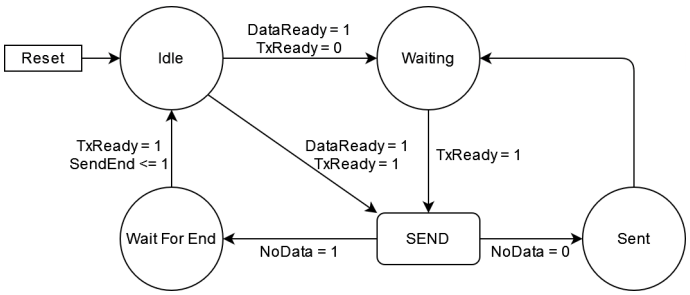

Figure 8: ASCII Sender Mealy State Machine Diagram

The sender sub-component starts after the number has been converted to digits or immediately upon receiving an invalid number. The sender also uses the digit registers and counter used by the converter. Operation of the sender is achieved using the state machine in Figure 8 as well as an ASCII converter which will take numerals and convert them to 8-bit ASCII characters. Once the data and the transmitter are ready, the SM will send a value, then it will wait for the transmitter to be ready again either to send the next numeral or, if there is no more data left, to send a carriage return signal before returning to idle. To send a value, the sender does the following: firstly, if the number is negative, it will tell the converter to output the ASCII code for a negative sign, number 45, then mark the fact a negative has been sent to avoid repeating this. Next, it will send the numeral stored in the register that the counter indexes to the converter. The converter will add 48 to the number to convert it to an 8-bit ASCII code. If the counter is now at zero, it means the number is done sending and the "end of data" boolean is set to true. If the counter is not zero, it is decremented by one. The exception to all this is if the invalid result input is given. In this case, the sender will output the invalid result character (! or ASCII 33) and the end of data variable is set to true. Then on the final send, when the end of data is true, it will output a carriage return and then signal to the state machine that the message is done.

4.4 Test Bench

The test bench for simulating the full calculator is identical to the test bench for the UART components. The test bench creates an instance of the top-level implementation in Figure 1 and then creates a second UART component with a different clock to transmit the input and decode the output. This second transmitter is once again given input data from a text file with hex codes.

Figure 9 shows the calculator computing 1 + 1. This Figure shows that most of the calculator's time is spent on receiving and sending, as each reception takes 82080ns and each transmission 86800ns. This matches with the theory as at a baud rate of 115.2kHz, each bit is asserted for 8680 ns. Each packet has 10 bits meaning a time of 86800 ns is to be expected. Reception is technically quicker by half a period as it finishes halfway through the stop bit. However, this time is lost again in the idle state. When receiving the inputs, the decoder takes only one clock cycle (20 ns) from when the receiver marks the data as ready to convert. Once the decoder is done, the calculator always takes 60 ns to compute a result, including invalid operations. However, this is not guaranteed and depends on the numeric_std library. Once the result is

ready, the encoder takes a variable amount of time to complete, with two clock cycles of fixed cost (40 ns) and an additional clock cycle per digit of the result. All these times are negligible, as even the largest operation would take only $20 + 60 + (40 + 5 * 20) = 220$ ns, which is less time than it takes for the start-bit of the final echoed character to send.

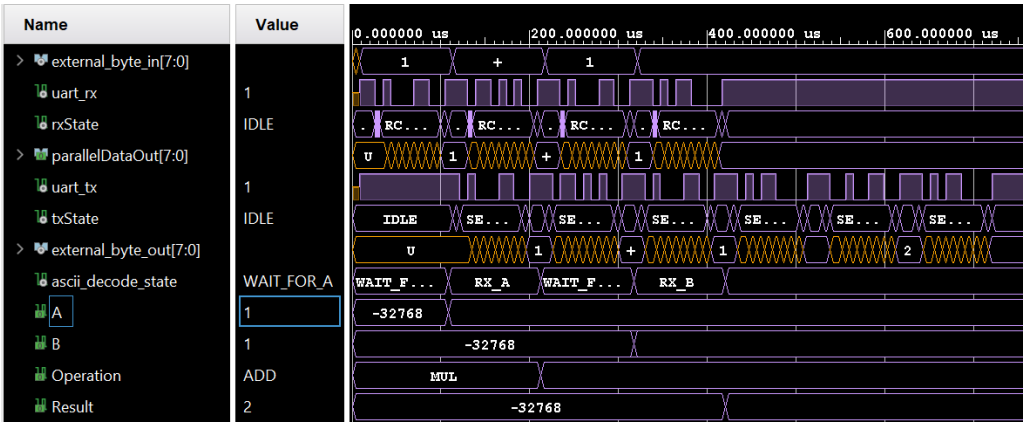

Figure 9: Simulation of 1 + 1

Unlike the multiplier assignment before, testing all possible inputs is nearly impossible, as there are 2^{16} possible values for A and B as well as the four operations, meaning that there are 2^{34} valid inputs. This does not include testing more inputs, such as injecting random characters or using more negative signs than necessary.

Therefore testing was done by having a test case for every possible case per component. For the receiver and transmitter, every relevant ASCII char was tested. This means all ten numerals, four operators, and the carriage return had to be received and transmitted once. For the decoder, every numeral was tested once (in various positions), as well as negative and positive numbers for all possible input lengths. Some exceptional test cases were also added, such as multiple negative signs and superfluous characters. The calculator was tested with every type of operation, alongside some purposeful errors. Finally, the encoder had to be tested with positive and negative results of all possible lengths. The appendix includes a series of operations that meet all of these conditions. These were converted to hex inputs and written to a text file used as the final simulation test for the design.

5 Implementation

Taking the calculator from design to running on physical hardware posed various challenges. The first step was converting the design to work within the provided code for implementation, such as setting up the proper connections and removing the now obsolete DIP switch connection. After this, various timing issues created problems for the design, all of which were flagged by Vivado. The first issue was the timing created by running some of the state machines on every clock. To fix this, the state machines' primary processes had their sensitivity list changed to only occur when inputs changed. A further refactor was considered to break the state machines into separate processes for the next-state logic, state memory, output logic, and output register. A refactor such as this would improve performance by having less logic in one process [4], however, due to time constraints, the decision was made to extend the calculator to 16-bit instead of doing this refactor. Another timing issue occurred in the calculator, with negative slack timing when dividing. The fix for this problem was adding a state machine to the calculator when earlier iterations were purely combinational. Even with these changes and an implementation with no error, the generated bit-stream did not create any output when loaded on the FPGA. This was also the case when the originally provided code was loaded onto the FPGA, implying the error may not be with the calculator design itself but instead some error during implementation. With more time, perhaps the FPGA could be programmed correctly.

A References

[1] E. Peňa and M. G. Legaspi. "Uart: A hardware communication protocol understanding universal asynchronous receiver/transmitter." (Apr. 8, 2023), [Online]. Available: https://www.analog.com/en/analog-dialogue/articles/uart-a-hardware-communication-protocol.html (visited on 04/08/2023).

[2] R. Keim. "Uart baud rate: How accurate does it need to be?" (Jan. 25, 2017), [Online]. Available: https://www.allaboutcircuits.com/technical-articles/the-uart-baud-rate-clock-how-accurate-does-it-need-to-be/ (visited on 04/08/2023).

[3] W. J. Dally, R. C. Harting, and T. M. Aamodt, *Digital Design using VHDL A Systems Approach*. Cambridge University Press, 2016.

[4] J. Wakerly, *Digital Design: Principles and Practices* (Prentice Hall Signal Processing Series). Prentice Hall, 1990, ISBN: 9780132128384. [Online]. Available: https://books.google.co.uk/books?id=1dfA6kNX7-UC.

B Appendix

B.1 Test Cases

$$0 + -67890 = -67890 \tag{1}$$
$$1 - -1234 = 1235 \tag{2}$$
$$23 * -567 = -13041 \tag{3}$$
$$456/-89 = -5 \tag{4}$$
$$--7890 + -1 = 7889 \tag{5}$$
$$12345/0 = \text{DIV ZERO} \tag{6}$$
$$-67890/6 = -11315 \tag{7}$$
$$-1234 + 23 = -1211 \tag{8}$$
$$-567 + --456 = -111 \tag{9}$$
$$---89 + 7890 = 7801 \tag{10}$$
$$-1 + 12345 = 12344 \tag{11}$$
$$4 * ---22 = -88 \tag{12}$$
$$123 - 123 = 0 \tag{13}$$
$$256/12 = 22 \tag{14}$$
$$128 + 128 = 256 \tag{15}$$
$$1 + 1 = 2 \tag{16}$$